YAACT: A genetic algorithm tool for code coverage analysis

Víctor Rodríguez
Jindrizka Dominguez
Gilberto Sánchez

April 28 2014

1 Introduction

Testing is both technically and economically an important part of high quality software production. It has been estimated that testing accounts for half of the expenses in software production. Much of the testing is done manually or using other labor-intensive methods. It is thus vital for the software industry to develop efficient, cost effective, and automatic means and tools for software testing. Researchers have proposed several methods over years to generate automatically solution which have different drawbacks. This study examines automatic software testing optimization by using genetic algorithm approaches. This study will cover two approaches: a) obtain the sequence of regression tests that cover the greatest amount of code and b) once it is achieved another genetic algorithm will eliminate tests cases that cover the same section of code on the basis of still get the maximum code coverage. The overall aim of this research is to reduce the number of test cases that need to be run with the greatest amount of code covered.

Regression testing is expensive but an essential activity in software maintenance. Regression testing attempts to validate modified software and ensure that the modified parts of the program do not introduce unexpected errors. It executes an existing test suite on a changed program to assure that the program is not adversely affected by unintended amendments[1]. The time used for regression testing can be assumed approximately half of the software maintenance activities. However, through the use of an effective prioritization sequence, testers can reorder(or eliminate) the test cases to reduce the time and cost in the system, allowing corrections to be made earlier and raising overall confidence that the software has been adequately tested. One concern in regression testing is the effectiveness of test suites in finding new faults in successive program versions. A regression test selection technique may help us to select an appropriate number of test cases from this test suite.

Many automatic test case generators have been developed and many classical searching methods have been used to derive test case. However, these techniques work only for continuous function (while most program domains are of discontinuous spaces). This is where Evolutionary Algorithms steps in, such as those using Genetic Algorithms GA. GAs are heuristic search techniques which are able to search a discontinuous space. Evolutionary Testing is a promising approach to entirely automate test case design. It can be used to generate test cases for structural testing. In some projects[2] [4] a GA based test data generation is presented. However, we will not afect the test data input or try to create an automatic generator of tests cases [3].

The proposed GA is applied to different types of software systems small programs with different complexity. Results compared to random testing showed that genetic algorithms can be used effectively in automatic software testing to generate test data for unit testing.

2 Algorithm

This study will apply genetic algorithms (GA) to solve two aproaches of testing automation:

1. GA to obtain the sequence of regression tests that cover the greatest amount of code (GA to increase code coverage).
2. GA to find the minimal secuence of regression tests that cover the greatest amount of code (GA to get minimal funtional test suite).

The following figure 1 and figure 4 show theese two main aproaches:

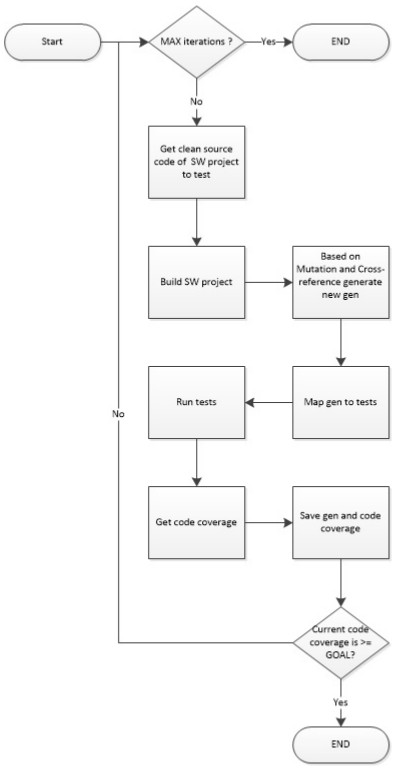
Start
MAX iterations ?
Yes
END
No
Get clean source code of SW project to test
Build SW project
Based on Mutation and Cross-reference generate new gen
Map gen to tests
Run tests
Get code coverage
Save gen and code coverage
Current code coverage is >= GOAL?
No
Yes
END

Figure 1: GA to increase code coverage

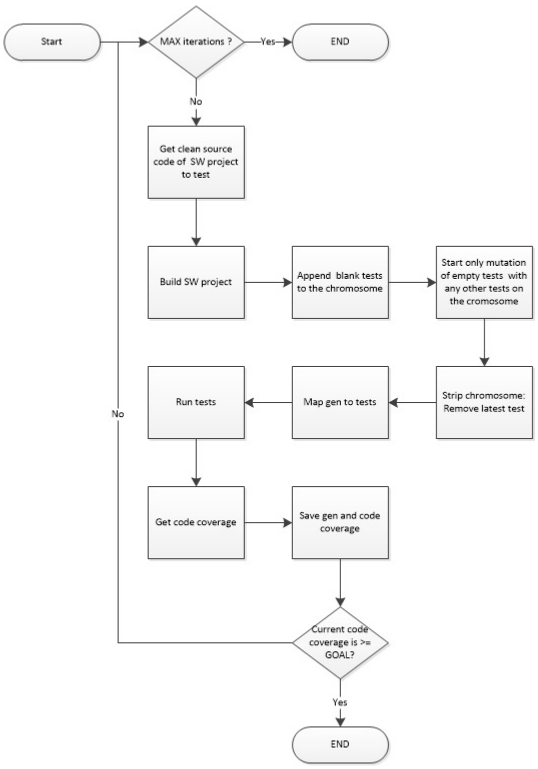
Start
MAX iterations ?
Yes
END
No
Get clean source code of SW project to test
Build SW project
Append blank tests to the chromosome
Start only mutation of empty tests with any other tests on the cromosome
Strip chromosome: Remove latest test
Map gen to tests
Run tests
No
Get code coverage
Save gen and code coverage
Current code coverage is >= GOAL?
Yes
END

Figure 2: GA to get minimal functional test suite

Before going furter we will describe the Genetic Algorithm in a very light way, in order to avoid knowledge gaps.

Genetic algorithm is a population-based search method. Genetic algorithms are acknowledged as good solvers for tough problems. Genetic algorithms require several parameters including the following[1]:

1. Maximum number of generations, G,
2. Population size, P,
3. Crossover rate, pc,
4. Mutation rate, pm,
5. Convergence criterion.

There are two main concepts in genetic algorithm: crossover and mutation.

Crossover : A binary variation operator is called recombination or crossover. This operator merges information from two parent genotypes into one or two offspring genotypes. Similarly to mutation, crossover is a stochastic operator: the choice of what parts of each parent are combined, and the way these parts are combined, depends on random drawings. The principle behind crossover is simple: by mating two individuals with different but desirable features, we can produce an offspring which combines both of those features.

```
   child_1 = parent_1(:size-1)/2)
\\ remove all duplicated genotypes from in parent_2
\\ that match genotypes found in child_1
   new_child = parent_2 + child_1
```

Mutation: A unary variation operator is called mutation. It is applied to one genotype and delivers a modified mutant, the child or offspring of it. In general, mutation is supposed to cause a random unbiased change. Mutation has a theoretical role: it can guarantee that the space is connected.

```
   child_value = parent[random_index_1]
\\ switch two genotypes randomly
   children_pop[random_index] = children_pop[random_index_2]
   children_pop[random_index_2] = child_value
```

The Figure 1 shows the algorithm for the first problem : GA to obtain the sequence of regression tests that cover the greatest amount of code (GA to increase code coverage). This algorithm will execute the following steps:

1. Check if we have reach the max number of iteration. If true stop the execution else continue with the execution of the algorithm.
2. Get clean source code of SW project to test.
3. Build SW project (for codecoverage).
4. Based on Mutation and Cross-reference generate new gen.
5. Map gen to tests: Map the chromosome numbers to specific tests.
6. Run tests.
7. Get code coverage percentage.
8. Save the chromosome and its code coverage into a history file. Also the previous code coverage and chromosome is saved into a history file.
9. Current code coverage is ¿= GOAL? . If true Stop execution else back to begining.

This algorithm try to generate a sequence of tests that maximaze the code coverage. The aproach will not be valid if we have duplicated tests. A duplicated functional tests will not increase the code coverage. The kind of tests that we will need to use are functional tests.

The Figure 4 shows the algorithm for the second problem: GA to find the minimal secuence of regression tests that cover the greatest amount of code (GA to get minimal funtional test suite). This algorithm will execute the following steps:

1. Check if we have reach the max number of iteration. If true stop the execution else continue with the execution of the algorithm

2. Get clean source code of SW project to test.
3. Build SW project (for codecoverage).
4. Append empty test to the chromosome.
5. Start only mutation of empty tests with any other tests on the chromosome.
6. Strip chromosome: Remove latest test.
7. Map gen to tests: Map the chromosome numbers to specific tests.
8. Run tests.
9. Get code coverage percentage.
10. Save the chromosome and its code coverage into a history file. Also the previous code coverage and chromosome is saved into a history file.
11. Current code coverage is ¿= GOAL?. If true Stop execution else back to begining.

This GA will try to minimize the number of tests that execute the same portion of the code. Thiw will reduce the total amount of time of the regression test suite.

The complete source code and history of development is detailed on the git repository:

`https://github.com/VictorRodriguez/yaact`

3 Results

After doing multiple iterations on the GA for optimizaton we realize that the code coverage was inotincreasing no matter how many iterations we could test our solution

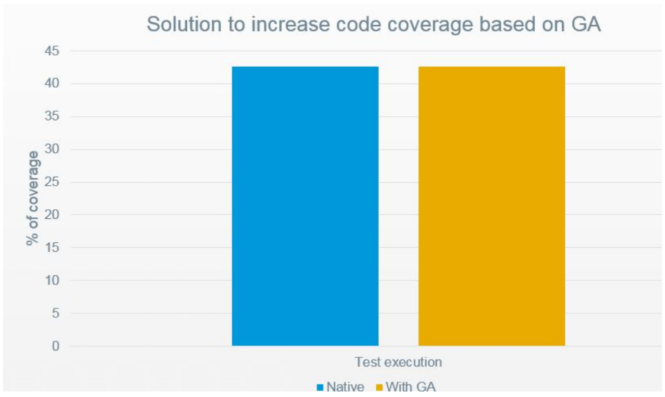

Figure 3: Solution A: GA to increase code coverage

The same problen hapend with the secind GA aproach. After many iteratiosn we stil get that the same number of test cases with maximum level of code covefage

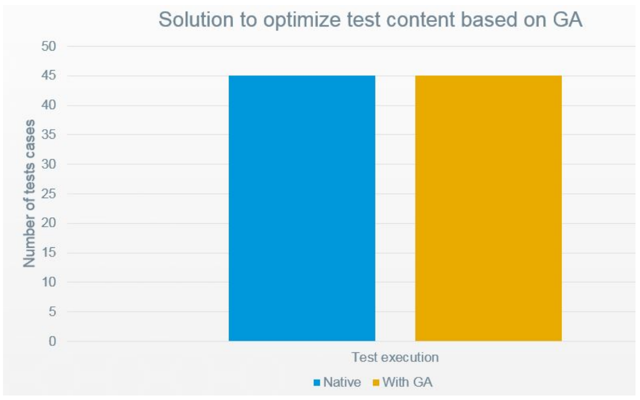

Figure 4: Solution B: GA to get minimal functional test suite

4 Conclusion

In Conclusion, genetic algorithms will not solve all testing needs in all systems.This solution does not apply for all SW products. It requires a level of complexity (still not defined) on the SW product and their integration/application tests suites. It was discovered that in order to see an increase in code coverage, of the test suite used, there must be a development of new test cases. This happens because each time tests are executed the same code is being exercised.

This approach will not affect the following tests suites :

1. Unit tests
2. Feature Test

This approach might affect the following tests suites :

1. Integration/compatibility tests
2. Performance benchmarks
3. Stress tests
4. Longevity tests

5 References

References

[1] Suman and Seema, A Genetic Algorithm for Regression Test Sequence Optimization, Yadavindra College of Engineering, International Journal of Advanced Research in Computer and Communication Engineering Vol. 1, Issue 7, September 2012

[2] Maha,study of genetic algorithm for automatic software test data generation,university of pune, giirj, vol.1 (2), december (2013)

[3] Cristian Cadar/Daniel Dunbar/ Dawson Engler, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,Stanford University

[4] Ganesh, EXE: Automatically generating inputs of death. ,Stanford University,In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006).